Functions

In this exercise you will learn how to write a FUNCTION subprogram, and how to use a COMMON block to share variables between subroutines, functions, and the main program. You will again simulate the motion of a projectile moving through the air with drag forces, but now with a test of the conservation of energy. You will also be introduced to the X11 windowing system.

1. Problem

This exercise and the exercise which follows it are actually two halves of one project. The particular problem we wish to solve is this: to display the trajectory of an object fired into the air from the ground (perhaps a baseball, a cannon ball, or a potato), when the object is subject to both gravity and air resistance. The equations for doing this are the same as those used in Exercise 10 to simulate a falling flour bomb; the only difference is in the initial conditions. A projectile fired upward at an angle will have an initial velocity in both the positive vertical and horizontal directions. Your first program (this exercise) should track the projectile as it rises and falls back to the ground, writing the x and y positions to an output data file, one line at a time for every second of the simulation. Your program for the following exercise will then read the data in this file into a pair of arrays, and then pass these arrays to a subroutine which will draw a curve of your data.

The advantage to solving this problem by writing two programs is that since your graphics program can read any data file you name, not just the particular data for one problem, it will be useful for a wider range of problems. This demonstrates the general idea of solving a problem by breaking it down into separate, general steps and then writing separate programs (or perhaps subroutines or functions) to accomplish these steps. In the present case you will have one program produce an output file which will then be an input file for the other program. For more complicated problems you might have several programs producing output files which are read as input by several other programs, or you could have several subroutines interacting with data passed as arguments or kept in COMMON storage, all to accomplish a specified set of goals.

Your program should use a FUNCTION to compute the new position of the projectile after a timestep of Δt. All calculations dealing with moving the projectile forward in time should be put into this function, not the main program. The timestep should be the only parameter passed to the function as an argument. The function should return the amount of energy lost as work done against the drag force (see below). This will give us a way to check the accuracy of the simulation by testing the conservation of energy. The vertical and horizontal position of the object should be passed back to the main program via a COMMON block. The components of the position and the velocity should also be kept in the COMMON block, both so that their values are preserved between calls to the function, and so that their values are available in the main program for debugging. You will probably find it useful to store other variables in COMMON so that they can be used in both the main program an in the function.

2. Work and Energy

In Exercise 10 your program computed the position of a projectile by applying Newton's second law ($F = ma$) over a small time step Δt. Now you will also want to keep track of the energy of the projectile.

In physics the definitions of "work" is that it is the product of the force applied in the direction of motion and the distance traveled. So for two dimensional motion the total work is the product of F_x, the x component of the force, and Δx, the x displacement during the step, added to the product of F_y, the y component of the force, and Δy, the y displacement during the step:

$$\Delta W = F_x \Delta x + F_y \Delta y$$

Your program should now compute ΔW for each small step and sum up the total work done during the time of flight of the projectile. This is energy lost by the projectile due to drag force.

Kinetic energy is defined as

$$K = \frac{1}{2} m v^2$$

where v is the magnitude of the velocity (don't forget both x and y components.) Gravitational potential energy is the work done against gravity in lifting an object a height y:

$$U = mgy$$

If there were no drag force then $K + U$ would be a constant, and we would say that total mechanical energy is "conserved." When there is drag then some energy is lost as work done by the drag force, so $K + U$ should decrease.

3. Input/Output

Your program should take as inputs the mass and surface area of your projectile (on one line, in that order, in grams and cm^2, respectively), followed by the initial velocity (in m/sec) and angle of elevation (in degrees) at which the projectile is fired (again, both on one line).

The program should output the x and y positions of the projectile, as one pair of numbers per line, until the object strikes the ground. Your program should only print the projectile position for every second of the simulation, not every time step. Your program will take many smaller time steps, but then output the position only every second of the simulation. The time should *not* be written to the file. The final position shold be written to the file even if it does not occur right on a given whole second.

The output should be written to a data file, not to the terminal. The data file should have a name composed of your userid and the the exercise number, followed by "`.d`" (meaning "data") as the file extension. Thus Matthew Vassar would write the program `mavassar14.f`, which produces the data file `mavassar14.d` (which will be read and plotted by the program `mavassar15.f` in the next exercise).

When your program is done it should print the maximum height attained by the projectile, the horizontal distance traveled, the time taken for the trip, and the components of the velocity and acceleration at impact. These should be written to the terminal, not the data file, in a neatly presented report. Optionally, your report can also include the details of the launch parameters (the input data used by your program).

Finally, your program should print the initial energy of the projectile (it's all kinetic energy), the final energy (again, at the ground it is all kinetic energy), and it should compare the difference between these two values, which is one way to compute the energy lost to the drag force, to the accumulated work done by the projectile as reported by your function. If your program is functioning correctly then these two different ways of computing the lost energy should agree fairly well. If they don't, something may be wrong.

4. Testing

To test your program with realistic data you should use an area corresponding to a diameter of 2 inches (i.e. a radius of 2.54 cm), a mass of 47 grams (a baseball), and an initial velocity of about 175 m/s. To study the effects of drag forces we should exagerate the effects by choosing a large coefficient of drag, so set $C_D = 1.5$ (but do not ask the user for a value).

Since you can compute exactly the time it takes an object to go up and come back down when there is no drag force, compare the results for the time of flight from your program with the exact zero-drag answer. You can trick your program into assuming that there is no drag force by telling it that the surface area is zero.

You may like to know that the sine and cosine functions in Fortran take their arguments in radians, not degrees.

You may want to check that your code from Exercise 10 correctly determines the direction of the drag force in both the vertical and horizontal directions. The drag force always operates in the *opposite* direction to the velocity. For the vertical motion this is no longer necessarily upward.

If you find that the two ways of computing the energy lost to work done by the drag force do not agree but your program appears to be working more or less correctly otherwise, then it is possible that your timestep is too large. Try a smaller timestep and see if you get better agreement. On the other hand, if the two methods agree well then try a larger timestep to speed up your code. In the end, you need to pick a timestep that is not too large and not too small.

5. Reading

You should read about FUNCTION subprograms, and about how variables may be shared between functions, subroutines, and the main program by using a COMMON block.

The values of variables in a function or subroutine may also be preserved using the SAVE command, so you may want to read about this alternative.

6. The X11 Windowing system

The windowing system used on almost all Unix systems is called "X", or sometimes X11 or X11R6, because the latest version is version 11, revision 6. X was originally developed at MIT, but it has always been available to the wider Unix community. The implementation of X11 used under Linux comes from the XFree86 project, which developed an open-source distribution of X11 for the Intel x86 architecture. X is also available for the MacOS X platform, and on Microsoft Windows. The version of X11 on MacOS X co-exists easily with the native Mac windowing system, which is called Aqua. (The use of X in "Mac OS X" means version 10 of Mac OS, and is not related to the X11 windowing system.)

One of the many useful features of X is that it can display windows across the network. Thus you can log in to a remote host, give a command to display some information, and it will appear automatically in a window on your local workstation. If you connect to the remote host via the Secure Shell '`ssh`' command then you can turn on such window forwarding by adding the "`-X`" flag to the command. Let us say that you are initially logged in on a workstation called "helios" and running X11 on that workstation, but that you want to do some work on a computer called "noether". Or you may be logged on to a Mac workstation called "fortuna" and have already started the X11 windowing system and started an `xterm` terminal shell window. Either way, you can then log in to noether with the command:

```
% ssh -X mavassar@noether.vassar.edu
```

An amusing way to check the window forwarding is to run the '`xeyes`' command, which just pops up on your display a pair of eyes that follow around the pointer as you move the mouse. The command is

```
% xeyes
```

You can end this program by pressing control-C. If the remote host does not have `xeyes` (noether does, but some don't) then you can use any other common X11 application, such as ‘`xterm`‘, which opens up a terminal window containing a command shell from the remote host.

To get ready for using X11 in the next exercise you should try to connect to noether using X11 using one of the computers in our computer room, the intro lab, or your own computer. Start a terminal “shell” using the ‘`xterm`‘ application, and try out ‘`xeyes`‘ or other X applications.

One Unix application you might find useful for testing your program is `xmgr` (also called Grace). If you say `‘xmgr datafile.d‘` while the X11 windowing system is operating it will plot a simple graph of the data in the file `datafile.d`.

7. Configuring ssh

It is possible to have ssh automatically forward X11 windows, but this is often not the default. You might try connecting with ssh but without the “`-X`” flag to see if `xeyes` or `xterm` still pop up on your screen. If so, fine. If not, and you want to make it the default, then you can do so by editing the file `ssh_config` in the directory `/etc/ssh` (or perhaps just `/etc` – that is where it's found on MacOS X) and uncommenting out the appropriate line and changing it to read

```
ForwardX11 yes
```

Of course you have to be the system administrator (“root”) on the local machine to make such changes.